Meta-descent for Online, Continual Prediction

Andrew Jacobsen,[1] Matthew Schlegel,[1] Cameron Linke,[1]
Thomas Degris,[2] Adam White, [1,3] Martha White [1]
[1]University of Alberta, Edmonton, Canada,
[2]Google DeepMind, London, UK
[3]Google DeepMind, Edmonton, Canada
ajjacobs@ualberta.ca, mkschleg@ualberta.ca, clinke@ualberta.ca
thomas.degris@gmail.com, amw8@ualberta.ca, whitem@ualberta.ca

Abstract

This paper investigates different vector step-size adaptation approaches for non-stationary online, continual prediction problems. Vanilla stochastic gradient descent can be considerably improved by scaling the update with a vector of appropriately chosen step-sizes. Many methods, including AdaGrad, RMSProp, and AMSGrad, keep statistics about the learning process to approximate a second order update—a vector approximation of the inverse Hessian. Another family of approaches use meta-gradient descent to adapt the step-size parameters to minimize prediction error. These meta-descent strategies are promising for non-stationary problems, but have not been as extensively explored as quasi-second order methods. We first derive a general, incremental meta-descent algorithm, called AdaGain, designed to be applicable to a much broader range of algorithms, including those with semi-gradient updates or even those with accelerations, such as RMSProp. We provide an empirical comparison of methods from both families. We conclude that methods from both families can perform well, but in non-stationary prediction problems the meta-descent methods exhibit advantages. Our method is particularly robust across several prediction problems, and is competitive with the state-of-the-art method on a large-scale, time-series prediction problem on real data from a mobile robot.

Introduction

In this paper we consider continual, non-stationary prediction problems. Consider a learning system whose objective is to learn a large collection of predictions about an agent's future interactions with the world. The predictions specify the value of some signal many steps in the future, given that the agent follows some specific course of action. There are many examples of such prediction learning systems including Predictive State Representations (Littman, Sutton, and Singh 2001), Observable Operator Models (Jaeger 2000), Temporal-difference Networks (Sutton and Tanner 2004), and General Value Functions (Sutton et al. 2011). In our setting, the agent continually interacts with the world, making new predictions about the future, and revising its previous predictions as new outcomes are revealed. Occasionally, partially due to changes in the world and partially due to changes in the agent's own behaviour, the targets may change and the agent must refine its predictions. [1]

Stochastic gradient descent (SGD) is a natural choice for our setting because gradient descent methods work well when paired with abundant training data. The performance of SGD is dependent on the step-size parameter (scalar, vector or matrix), which scales the gradient to mitigate sample variance and improve data efficiency. Most modern large-scale learning systems make use of optimization algorithms that attempt to approximate stochastic second-order gradient descent to adjust both the direction and magnitude of the descent direction, with early work indicating the benefits of such quasi-second order methods if used carefully in the stochastic case (Schraudolph, Yu, and Günter 2007; Bordes, Bottou, and Gallinari 2009). Many of these algorithms attempt to approximate the diagonal of the inverse Hessian, which describes the curvature of the loss function, and so maintain a vector of step-sizes—one for each parameter. Starting from AdaGrad (McMahan and Streeter 2010; Duchi, Hazan, and Singer 2011), several diagonal approximations have been proposed, including RmsProp (Tieleman and Hinton 2012), AdaDelta (Zeiler 2012), vSGD (Schaul, Zhang, and LeCun 2013), Adam (Kingma and Ba 2015) and AmsGrad (Reddi, Kale, and Kumar 2018). Stochastic quasi-second order updates have been derived specifically for temporal difference learning, with some empirical success (Meyer et al. 2014), particularly in terms of parameter sensitivity (Pan, White, and White 2017; Pan, Azer, and White 2017). On the other hand, second order methods, by design, assume the loss and thus Hessian are fixed, and so non-stationary dynamics or drifting targets could be problematic.

A related family of optimization algorithms, called *meta-descent* algorithms, were developed for continual, online prediction problems. These algorithms perform meta-gradient descent adapting a vector of step-size parameters to minimize the error of the base learner, instead of approx-

[1]We exclude recent meta-learning frameworks (MAML (Finn, Abbeel, and Levine 2017), LTLGDGD (Andrychowicz et al. 2016)) because they assume access to a collection of tasks that can be sampled independently, enabling the agent to learn how to select meta-parameters for a new problem. In our setting, the agent must solve a large collection of non-stationary prediction problems in parallel using off-policy learning methods.

imating the Hessian. Meta-descent applied to the step-size was first introduced for online least-mean squares methods (Jacobs 1988; Sutton 1992b; 1992a; Almeida et al. 1998; Mahmood et al. 2012), including the linear complexity method IDBD (Sutton 1992b). IDBD was later extended to more general losses (Schraudolph 1999) and to support (semi-gradient) temporal difference methods (Dabney and Barto 2012; Dabney 2014; Kearney et al. 2018). These methods are well-suited to non-stationary problems, and have been shown to ignore irrelevant features. The main limitation of several of these meta-descent algorithms, however, is that the derivations are heuristic, making it difficult to extend to new settings beyond linear temporal difference learning. The more general approaches, like Stochastic Meta-Descent (SMD) (Schraudolph 1999), require the update to be a stochastic gradient descent update and have some issues in biasing towards smaller step-sizes (Wu et al. 2018). It remains an open challenge to make these meta-descent strategies as broadly and easily applicable as the AdaGrad variants.

In this paper we introduce a new meta-descent algorithm, called AdaGain, that attempts to optimize the stability of the base learner, rather than convergence to a fixed point. AdaGain is built on a generic derivation scheme that allows it to be easily combined with a variety of base-learners including SGD, (semi-gradient) temporal-difference learning and even optimized SGD updates, like AMSGrad. Our goal is to investigate the utility of both meta-descent methods and the more widely used quasi-second order optimizers in online, continual prediction problems.We provide an extensive empirical comparison on (1) canonical optimization problems that are difficult to optimize with large flat regions (2) an online, supervised tracking problem where the optimal step-sizes can be computed, (3) a finite Markov Decision Process with linear features that cause conventional temporal difference learning to diverge, and (4) a high-dimensional time-series prediction problem using data generated from a real mobile robot. In problems with non-stationary dynamics the meta-descent methods can exhibit an advantage over the quasi-second order methods. On the difficult optimization problems, however, meta-descent methods fail, which, retrospectively, is unsurprising given the meta-optimization problem for stepsizes is similarly difficult to optimize. We show that AdaGain can possess the advantages of both families — performing well on both optimization problems with flat regions as well as non-stationary problems — by selecting an appropriate base learner, such as RMSProp.

Background and Notation

In this paper we consider *online continual prediction* problems modeled as non-stationary, uncontrolled dynamical systems. On each discrete time step t, the agent observes the internal state of the system through an imperfect summary vector $\mathbf{o}_t \in \mathcal{O} \in \mathbb{R}^d$ for some $d \in \mathbb{N}$, such as the sensor readings of a mobile robot. On each step, the agent makes a prediction about a target signal $T_t \in \mathbb{R}$. In the simplest case, the target of the prediction is a component i of the observation vector on the next step $T_t = \mathbf{o}_{t+1,i}$—the classic one-step prediction. In the more general case, the target is constructed by mapping the entire future of the observation time series to a scalar, such as the discounted sum formulation used in reinforcement learning: $T_t = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k \mathbf{o}_{t+k+1,i}]$, where $\gamma \in [0, 1)$ discounts the contribution of future observations to the infinite sum. The prediction $P_t \in \mathbb{R}$ is generated by a parametrized function, with modifiable parameter vector $\mathbf{w}_t \in \mathbb{R}^k$.

In online continual prediction problems the agent updates its predictions (via $\mathbf{w}_t$) with each new sample $\mathbf{o}_t$, unlike the more common batch and stochastic settings. The agent's objective is to minimize the error between the prediction P_t given by $\mathbf{w}_t$ and the target T_t before it is observed, over all time steps. Online continual prediction problems are typically solved using stochastic updates to adapt the parameter vector $\mathbf{w}_t$ after each time step t to reduce the error (retroactively) between P_t and T_t. Generically, for stochastic *update vector* $\Delta_t \in \mathbb{R}^d$, the weights are modified

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \boldsymbol{\alpha}_t \circ \Delta_t \tag{1}$$

for a vector step-size $\boldsymbol{\alpha}_t$, where the operator $\circ$ denotes element-wise multiplication. Given an update vector, the goal is to select $\boldsymbol{\alpha}_t$ to reduce error, into the future. Semi-gradient methods like temporal difference learning follow a similar scheme, but Δ_t is not the gradient of an objective function.

Step-size adaptation for the stationary setting is often based on estimating second-order updates.[2] The idea is to estimate the loss function $\ell : \mathbb{R}^d \to \mathbb{R}$ locally around the current weights $\mathbf{w}_t$ using a second-order Taylor series approximation—which requires the Hessian $\mathbf{H}_t$. A closed-form solution can then be obtained for the approximation, because it is a quadratic function, giving the next candidate solution $\mathbf{w}_{t+1} = \mathbf{w}_t - (\mathbf{H}_t)^{-1} \nabla \ell(\mathbf{w}_t)$. If instead the Hessian is approximated—such as with a diagonal approximation—then we obtain *quasi-second order* updates. Taken to the extreme, with the Hessian approximated by a scalar, as $\mathbf{H}_t = \alpha_t^{-1}\mathbf{I}$, we obtain first-order gradient descent with a step-size of α_t. For the batch setting, the gains from second order methods are clear, with a convergence rate[3] of $O(1/t^2)$, as opposed to $O(1/t)$ for first-order descent.

These gains are not as clear in the stochastic setting, but diagonal approximations appear to provide an effective balance between computation and convergence rate improvements (Bordes, Bottou, and Gallinari 2009). Duchi, Hazan,

[2] A related class of algorithms are natural gradient methods, which aim to be robust to the functional parametrization. Incremental natural gradient methods have been proposed (Amari, Park, and Fukumizu 2000), including for policy evaluation with gradient TD methods (Dabney and Thomas 2014). However, these algorithms do not remove the need select a step-size, and so we do not consider them further here.

[3] There is a large literature on accelerated first-order descent methods, starting from early work on momentum (Nesterov 1983) and many since focused mainly on variance reduction (c.f. (Roux, Schmidt, and Bach 2012)). These methods can complement step-size adaptation, but are not well-suited to non-stationary problems because many of the algorithms are designed for a batch of data and focus on increasing convergence rate to a fixed minimum.

and Singer (2011) provide a general regret analysis for diagonal approximations methods proving sublinear regret if step-sizes decrease to zero overtime. One algorithm, AdaGrad, uses the vector step-size $\boldsymbol{\alpha}_t = \eta(\sum_{i=1}^{t} \Delta_t + \epsilon)^{-1}$ for a fixed $\eta > 0$ and a small $\epsilon > 0$, with element-wise division. RMSProp and Adam—which are not guaranteed to obtain sublinear regret—use a running average rather than a sum of gradients, with Adam additionally including a momentum term for faster convergence. AMSGrad is a modification of Adam, that satisfies the regret criteria, without decaying the step-sizes as aggressively as AdaGrad.

The *meta-descent* strategies instead directly learn step-sizes that minimize the same objective as the base learner. A simpler set of such methods, called *hypergradient* methods (Jacobs 1988; Almeida et al. 1998; Baydin et al. 2018), only adjust the step-size based on its impact on the weights on a single step. Hypergradient Descent (HD) (Baydin et al. 2018) takes the gradient of the loss $\ell(\mathbf{w})$ w.r.t. a scalar step-size $\alpha > 0$, to get the meta-gradient for the step-size as $\partial \ell(\mathbf{w}_t)/\partial \alpha = -\nabla_{\mathbf{w}} \ell(\mathbf{w}_{t-1})^\top \nabla_{\mathbf{w}} \ell(\mathbf{w}_t)$. The update simply requires storing the vector $\mathbf{g}_{t-1} = \nabla_{\mathbf{w}} \ell(\mathbf{w}_{t-1})$ and updating $\alpha_{t+1} = \alpha_t + \bar{\alpha} \mathbf{g}_{t-1}^\top \mathbf{g}_t$, for a meta step-size $\bar{\alpha} > 0$. More generally, meta-descent methods, like IDBD (Sutton 1992b) and SMD (Schraudolph 1999), consider the impact of the step-size back in time, through the weights, with $w_{t,j}$ the j-th element in vector $\mathbf{w}_t$

$$\frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial \alpha_i} = \sum_{j}^{k} \frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}} \frac{\partial w_{t,j}}{\partial \alpha_i}. \tag{2}$$

The goal is to approximate this gradient efficiently, usually using a recursive strategy. We derive such a strategy for AdaGain below using a different meta-descent objective, and for completeness include the derivation for the SMD objective in the appendix (as the original contains an error).

Illustrative example

To make the problem more concrete, consider a simple stateless tracking problem driven by two interacting Gaussians:

$$Y_t \stackrel{\text{def}}{=} Z_t + \mathcal{N}(0, \sigma_{Y,t}^2), \quad Z_{t+1} \leftarrow Z_t + \mathcal{N}(0, \sigma_{Z,t}^2). \tag{3}$$

where the agent only observes the sequence $Y_1, Y_2, \ldots$. The objective is minimize mean squared error (MSE) between a scalar prediction $P_t = w_t$ and the target $T_t = Y_{t+1}$. This problem is non-stationary because $\sigma_{Y,t}$ and $\sigma_{Z,t}$ change periodically and the agent has no knowledge of the schedule. Since $\sigma_{Y,t}$ and $\sigma_{Z,t}$ govern how quickly the mean Z_t drifts and the sampling variance in Y_t, the agent must step its step-size accordingly: larger $\sigma_{Z,t}$ requires larger stepsize, larger $\sigma_{Y,t}$ requires a smaller step-size. The agent must continually change its scalar step-size value in order to achieve low MSE. The optimal constant scalar step-size can be computed in this simple domain (Sutton 1992b), and is shown by the black dashed line in Figure 1. We compared the step-sizes learned by several well-know quasi-second order methods (AdaGrad, RMSProp, Adadelta) and three meta-descent strategies including our own AdaGain. We ran the experiment for over 24 hours to test the robustness of these methods in a long-running continual prediction task. Several methods including AdaGain were able to match the optimal step-size. However, several well-known methods including AdaGrad and AdaDelta completely fail in this problem. In addition, the meta-descent strategy SMD diverged after 8183817 time steps, highlighting the special challenges of online, continual prediction problems.

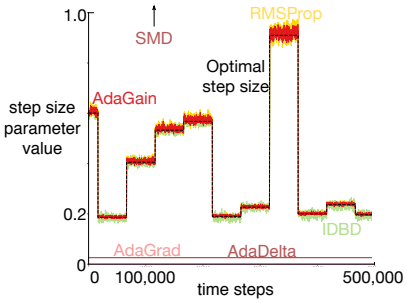

Figure 1: Optimal Gain Experiment. Depicted is the last 500,000 steps out of $3 * (10^9)$. AdaGrad, and AdaDelta fail to learn the correct progression of stepsizes, and SMD diverges.

Adaptive Gain for Stability

Tracking—continually updating the weights with recent experience—contrasts the typical goal of convergence. Much of the previous algorithm development for step-size adaptation, however, has been towards the aim of convergence, with algorithms like AdaGrad and AMSGrad that decay step-sizes over time. Assuming finite representational capacity, there may be aspects of the problem that can never be accurately modeled or predicted by the agent. In these partially observable problems tracking and thus treating the problem as if it were non-stationary can improve prediction accuracy compared with methods that converge (Sutton, Koop, and Silver 2007). In continual learning we assume the agent's task partially observable in this way, and develop a new step-size method that can facilitate tracking.

We treat the learning system as a dynamical system—where the weight update is based on stochastic updates known to suitably track the targets—and consider the choice of step-size as the inputs to the system to maintain *stability*. Such a view has been previously considered under adaptive gain for least-mean squares (LMS) (Benveniste, Metivier, and Priouret 1990, Chapter 4), where weights are treated as state following a random drift. To generalize this idea to other incremental algorithms, we propose a more general criteria based on the magnitude of the update vector.

A criteria for $\boldsymbol{\alpha}$ to maintain stability in the system is to keep the norm of the update vector small

$$\min_{\boldsymbol{\alpha} > 0} \mathbb{E}\left[\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2 \;\middle|\; \mathbf{w}_0 \right]. \tag{4}$$

The update $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ on this time step is dependent on the step-size $\boldsymbol{\alpha}$ because that step-size influences $\mathbf{w}_t$ and past updates. The expected value is over all possible update vectors $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ for the given step-size and assuming the

system started with some $\mathbf{w}_0$. If the dynamics are ergodic, $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ does not depend on the initial $\mathbf{w}_0$, and is only driven by the underlying state dynamics and the choice of $\boldsymbol{\alpha}$. The step-size can be seen as a control input for this system, with the goal to maintain a stable dynamical system by minimizing $\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2$ over time.

We derive an algorithm to estimate $\boldsymbol{\alpha}$ for this dynamical system, which we call AdaGain: Adaptive Gain for Stability. The algorithm is derived for a generic update $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ that is differentiable w.r.t. the weights $\mathbf{w}_t$; we provide specific examples for particular updates in the appendix, including for linear TD.

Generic algorithm with quadratic-complexity

We derive the full quadratic-complexity algorithm to start, and then introduce approximations to obtain a linear-complexity algorithm. To minimize (4), we use stochastic gradient descent, and thus need to compute the gradient of $\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2$ w.r.t. the step-size $\boldsymbol{\alpha}$. For step-size α_i as the ith element in the vector $\boldsymbol{\alpha}$, and $w_{t,j}$ the j-th element in vector $\mathbf{w}_t$

$$\begin{aligned}\frac{\frac{1}{2}\partial\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2}{\partial\alpha_i} &= \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))^\top \frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial\alpha_i} \\ &= \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))^\top \sum_j^k \frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}} \frac{\partial w_{t,j}}{\partial\alpha_i}.\end{aligned}$$

The key, then, is to track how a change in the weights impacts the update and how changes in the step-size impact the weights. The first term can be computed instantaneously on this step. For the second term, however, the impact of the step-size on the weights goes back further to previous updates. We show how to obtain a recursive form for this step-size gradient, $\boldsymbol{\psi}_{t,i} \stackrel{\text{def}}{=} \frac{\partial\mathbf{w}_t}{\partial\alpha_i} \in \mathbb{R}^k$.

$$\begin{aligned}\boldsymbol{\psi}_{t+1,i} &= \frac{\partial(\mathbf{w}_t + \boldsymbol{\alpha}\circ\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha})))}{\partial\alpha_i} \\ &= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha}\circ\sum_j \frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}}\frac{\partial w_{t,j}}{\partial\alpha_i} + \begin{bmatrix}\mathbf{0}\\ \Delta_{t,i}(\boldsymbol{\alpha})\\ \mathbf{0}\end{bmatrix} \\ &= (\mathbf{I} + \text{diag}(\boldsymbol{\alpha})\mathbf{G}_t)\boldsymbol{\psi}_{t,i} + \begin{bmatrix}\mathbf{0}\\ \Delta_{t,i}(\boldsymbol{\alpha})\\ \mathbf{0}\end{bmatrix},\end{aligned}$$

where $\mathbf{G}_{t,j} \stackrel{\text{def}}{=} \frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}} \in \mathbb{R}^k$, $\mathbf{G}_t \stackrel{\text{def}}{=} [\mathbf{G}_{t,1},\ldots,\mathbf{G}_{t,k}] \in \mathbb{R}^{k\times k}$, and Therefore, $\boldsymbol{\psi}_{t+1,i}$ represents a sum of updates, with a recursive weighting on previous $\boldsymbol{\psi}_{t,i}$ adjusting the weight of previous updates in the sum.

We can approximate the gradient using this recursive relationship, without storing all previous samples. Though the above updates are exact, we obtain an approximation when implementing such a recursive form in practice. When using $\boldsymbol{\psi}_{t-1,i}$ computed on the last time step $t-1$, this gradient estimate is in fact w.r.t. the previous step-size $\boldsymbol{\alpha}_{t-2}$, rather than $\boldsymbol{\alpha}_{t-1}$. Because these step-sizes are slowly changing, this gradient still provides a reasonable estimate; however, for many steps into the past, the accumulated gradients in $\boldsymbol{\psi}_{t,i}$ are likely inaccurate. To improve the approximation, and forget old gradients, we introduce a forgetting parameter $0<\beta<1$, which focuses the accumulation of gradients in $\boldsymbol{\psi}_{t,i}$ to a more recent window.

The gradient update to the step-size also needs to ensure that the step-sizes remain positive. Similarly to IDBD, we use an exponential form for the step-size, where $\alpha = \exp(\beta)$ and $\beta\in\mathbb{R}$ is updated with (unconstrained) stochastic gradient descent. Conveniently, as we show in the appendix, we do not need to maintain this auxiliary variable, and can simply directly update $\boldsymbol{\alpha}$.

The resulting generic updates for quadratic-complexity AdaGain, with meta step-size $\bar{\alpha}$, are

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1}\circ\exp\left(-\bar{\alpha}\boldsymbol{\alpha}_{t-1}\circ(\boldsymbol{\Psi}_t^\top\mathbf{G}_t^\top\Delta_t)\right) \tag{5}$$

$$\boldsymbol{\psi}_{t+1,i} = (1-\beta)\boldsymbol{\psi}_{t,i} + \beta\boldsymbol{\alpha}_t\circ(\mathbf{G}_t\boldsymbol{\psi}_{t,i}) + \beta\begin{bmatrix}\mathbf{0}\\ \Delta_{t,i}\\ \mathbf{0}\end{bmatrix}$$

where the exponential is applied element-wise, $\boldsymbol{\psi}_{0,i} = \mathbf{0}$, $\boldsymbol{\alpha}_0 = 0.1$ (or some initial value), and $(\boldsymbol{\Psi}_t)_{:,i} = \boldsymbol{\psi}_{t,i}$ with $\boldsymbol{\Psi}_t\in\mathbb{R}^{k\times k}$. For computational efficiency to avoid matrix-matrix multiplication, the order of multiplication for $\boldsymbol{\Psi}_t^\top\mathbf{G}_t^\top\Delta_t$ should start from the right, as $\boldsymbol{\Psi}_t^\top(\mathbf{G}_t^\top\Delta_t)$. The key complexity in deriving an AdaGain update, then, is simply in computing the Jacobian $\mathbf{G}_t$; given this, the remainder of the algorithm is fixed. For each update $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$, the Jacobian will be different, but is straightforward to compute.

Generic AdaGain algorithm with linear-complexity

Maintaining the entire matrix $\boldsymbol{\Psi}_t$ can be prohibitively expensive. As was done in IDBD (Sutton 1992b), one way to avoid maintaining this matrix is to assume that $\frac{\partial w_{t,j}}{\partial\alpha_i} = 0$ for $i\neq j$. This heuristic reflects that α_i is likely to have the largest impact on $w_{t,i}$, and less impact on the other entries in $\mathbf{w}_t$.

The modification above for this heuristic is straightforward, simply by setting entries $(\boldsymbol{\psi}_{t,i})_j = 0$ for $i\neq j$. This results in the simplification

$$\begin{aligned}\boldsymbol{\psi}_{t+1,i} &= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha}\circ\sum_j^k \mathbf{G}_{t,j}(\boldsymbol{\psi}_{t,i})_j + \begin{bmatrix}\mathbf{0}\\ \Delta_{t,i}(\boldsymbol{\alpha})\\ \mathbf{0}\end{bmatrix} \\ &= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha}\circ\mathbf{G}_{t,i}(\boldsymbol{\psi}_{t,i})_i + \begin{bmatrix}\mathbf{0}\\ \Delta_{t,i}(\boldsymbol{\alpha})\\ \mathbf{0}\end{bmatrix}.\end{aligned}$$

Further, since we will then assume that $(\boldsymbol{\psi}_{t+1,i})_j = 0$ for $i\neq j$, there is no purpose in computing the full vector $\mathbf{G}_{t,i}(\boldsymbol{\psi}_{t,i})_i$. Instead, we only need to compute the ith entry, i.e., for $\frac{\partial\Delta_{t,i}(\boldsymbol{\alpha})}{\partial w_{t,i}}$. We can then instead define $\hat{\psi}_{t,i}$ to be a scalar approximating $\frac{\partial w_{t,i}}{\partial\alpha_i}$, with $\hat{\boldsymbol{\psi}}_t$ the vector of these, and $\hat{\mathbf{j}}_t \stackrel{\text{def}}{=} \left[\frac{\partial\Delta_{t,1}(\boldsymbol{\alpha})}{\partial\mathbf{w}_{t,1}},\ldots,\frac{\partial\Delta_{t,k}(\boldsymbol{\alpha})}{\partial\mathbf{w}_{t,k}}\right]$ to define the recursion as $\hat{\boldsymbol{\psi}}_{t+1} \stackrel{\text{def}}{=} \hat{\boldsymbol{\psi}}_t + \boldsymbol{\alpha}\circ\hat{\mathbf{j}}_t\circ\hat{\boldsymbol{\psi}}_t + \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$, with $\hat{\boldsymbol{\psi}}_0 = \mathbf{0}$. The gradient using this approximation, with off-diagonals zero, is

$$\begin{aligned}\frac{\frac{1}{2}\partial\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2}{\partial\alpha_i} &= \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))^\top\sum_j^k\frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}}\frac{\partial w_{t,j}}{\partial\alpha_i} \\ &\approx \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))^\top\frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,i}}\frac{\partial w_{t,i}}{\partial\alpha_i} \\ &= \hat{\psi}_{t,i}\mathbf{G}_{t,i}^\top\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\end{aligned}$$

To compute this approximation, for all i, we still need to be able to compute $\mathbf{G}_t^\top \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$. In some cases this is straightforward, as is the case for linear TD (found in the appendix). More generally, we can use R-operators (Pearlmutter 1994) to compute this Jacobian-vector product, or a simple finite difference approximation, as we do in the appendix. Therefore, because we can compute this Jacobian-vector product in linear time, the only approximation is to $\hat{\boldsymbol{\psi}}_t$. The update is

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \exp\left(-\bar{\alpha}\, \boldsymbol{\alpha}_{t-1} \circ \hat{\boldsymbol{\psi}}_t \circ (\mathbf{G}_t^\top \Delta_t)\right) \quad (6)$$

$$\hat{\boldsymbol{\psi}}_{t+1} = (1-\beta)\hat{\boldsymbol{\psi}}_t + \beta \boldsymbol{\alpha}_t \circ \hat{\mathbf{j}}_t \circ \hat{\boldsymbol{\psi}}_t + \beta \Delta_t.$$

These approximations parallel diagonal approximations, for second-order techniques, which similarly assume off-diagonal elements are zero. Further, $\mathbf{G}_t$ itself is a gradient of the update w.r.t. the weights, where this update was already likely the gradient of the loss w.r.t. the weights. This $\mathbf{G}_t$, therefore, contains similar information as the Hessian. The AdaGain update, therefore, contains some information about curvature, but allows for updates that are not necessarily (true) gradient updates.

This AdaGain update is generic, but does require computing the Jacobian of a given update, which could be onerous in certain settings. We provide an update, based on finite differences in the appendix, that only requires differences between updates, that we have found works well in practice.

Experiments in synthetic tasks

We conduct experiments in several simulation domains to highlight the performance characteristics of meta-descent and quasi-second order methods. In our first experiment we investigate AdaGain and several meta-descent and quasi-second order approaches on a notoriously difficult stationary optimization task. Next we return to the simple state-less tracking problem described in the introduction, and investigate the parameter sensitivity of each method. Our third experiment investigates how different optimization algorithms can stabilize the iterates in sequential off-policy learning problems, which cause SGD-based methods to diverge. We conclude with a comparison of AdaGain and AMSGrad (the best performing quasi-second order method in the first three experiments) for online prediction on data generated by a mobile robot.

In all the experiments, we use AdaGain layered on-top of an RMSProp update, rather than a vanilla SGD update. As motivated earlier, meta-descent methods are not robust on difficult optimization surfaces, such as with flat or sharp regions. AdaGain provides a practical method to pursue meta-descent strategies that are robust to such realistic optimization problems. We motivate the importance of this choice in our first experiment on a difficult optimization task.

Function optimization. The aim of our first experiment is to investigate how AdaGain performs on optimization problems designed to be difficult for gradient descent. The Rosenbrock function is a two dimensional non-convex function, and the minimum is inside a flat parabolic shaped valley. We compared AMSGrad, SGD, and SMD, in each case extensively searching the meta-parameters of each method, averaging performance over 100 runs and 6000 optimization steps. The results are summarized in Figure 2, with trajectory plots of a single run of each algorithm, and the learning curves for all methods. AdaGain both learns faster and gets closer to the global optimum than all other methods considered. Further, two meta-descent methods, SMD and AdaGain without RMSProp perform poorly. This result highlights issues with applying meta-descent approaches without considering the optimization surface, and the importance of having an algorithm like AdaGain which can be combined with quasi-second order methods.

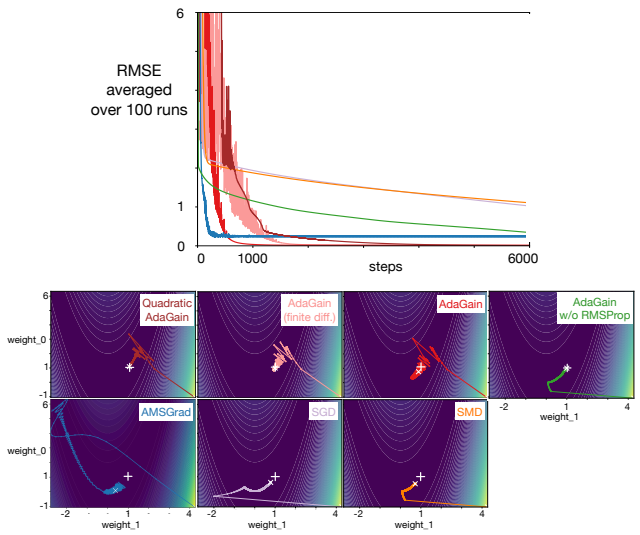

Figure 2: Optimization paths of a single run (with tuned meta-parameters) for several algorithms on the Rosenbrock function. The white $\times$ symbol indicates where in the input space the algorithm converged. The paths represent how each algorithm changes the weights while searching for the minimum. The white $+$ symbol indicates the optimal value for the weights—if $\times$ and $+$ symbol overlap the algorithm has reached the global minimum of the function. Although SGD and SMD appear to quickly approach the minimum, the valley is in fact easy to find, but reaching the $+$ is difficult. Neither method achieves a low final value, and converge slowly. The AdaGain algorithms with RMSProp—including full quadratic AdaGain algorithm, AdaGain with the linear approximation and AdaGain with the linear approximation and finite differences—outperform the other methods in this problem. The finite differences AdaGain algorithm is a generic strategy, that does not require knowledge of the Jacobian, and so can be easily applied to any updates (provided in the appendix). This result highlights that there is not a significant loss in using this approximation, over AdaGain with analytic Jacobians. AdaGain without RMSProp, on the other hand, converges much more slowly, though interestingly it does still outperform SMD. Note although the run above of AdaGain without RMSProp did reach the minimum, that was not true in general as reflected by the learning curve.

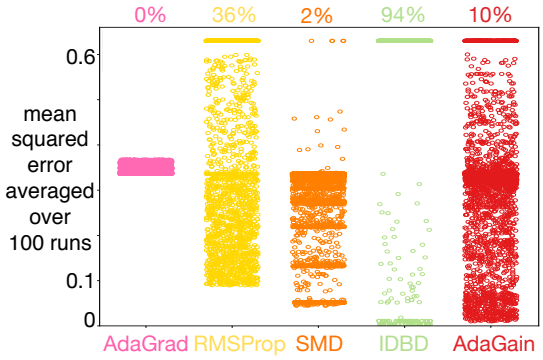

Figure 3: Parameter sensitivity plot for the first 500,000 steps of the stateless tracking problem. Each circle denotes the average MSE for a single parameter combination of an algorithm. The parameter combinations and respective performance are grouped in vertical columns for each method. The circles in each column are randomly offset within the column horizontally as many parameter settings may achieve almost identical MSE. Circles near the bottom of the plot represent low MSE. Circles arranged in a line in the top-most part of the plot are parameter combinations that either diverged or exceeded a minimum performance threshold, with the percentage of such parameter combinations given in the graph.

Stateless tracking problem. Recall from Figure 1, that several methods performed well in the stateless tracking problem; sensitivity to parameter settings, however, is also important. To help better understand these methods, we constructed a parameter sensitivity graph (Figure 3). IDBD can outperform AdaGain on this problem (lower MSE), but only a tiny fraction of IDBD's parameter settings achieve good performance. None of AdaGrad's parameter combinations exceeded the threshold, but all combinations resulted in high error compared with AdaGain. Many of the parameter combinations allowed AdaGain to achieve low error, suggesting AdaGain with a simple manual parameter tuning is likely to achieve good performance on this problem, while IDBD likely requires a comprehensive parameter sweep.

Baird's counterexample. Our final synthetic-domain experiment tests the stability of AdaGain's update when combined with the TD(λ) algorithm for off-policy state-value prediction in a Markov Decision Process. We use Baird's counterexample, which causes the weight's learned by off-policy TD(λ) (Sutton and Barto 1998) to diverge if a global step-size parameter is used (decaying or otherwise) (Baird 1995; Sutton and Barto 1998; Maei 2011). The key challenge is the feature representation, and the difference between the target and behavior policies. There is a shared redundant feature, and the weight associated seventh feature is initialized to a high value. The target policy always chooses to go to state seven and stay there forever. The behavior policy, on the other hand, only visits state seven 1/7 the time, causing large importance sampling corrections.

We applied AdaGain, AMSGrad, RMSprop, SMD, and TIDBD(Kearney et al. 2018)—a recent extension of the IDBD algorithm — to adapt the step-sizes of linear TD(λ) on Baird's counterexample. As before, the meta-parameters were extensively swept and the best performing parameters were used to generate the results for comparison. Figure 5 shows the learning curves of each method. Only AdaGain and AMSGrad are able to prevent divergence. SMD's performance is typical of Baird's counterexample: the meta-parameter search simply found parameters that caused extremely slow divergence. AdaGain learns significantly faster than AMSGrad, and achieves lower error.

To understand how AdaGain prevents divergence consider Figure 4. The left graph shows the step-size values as they evolve over time, and the right graph shows the corresponding weights. Recall, the weight for feature seven is initialized to a high value. AdaGain initially increases feature seven's step-size causing weight seven to quickly fall. In parallel AdaGain reduces the step-size for the redundant feature, preventing incorrect generalization. Over time the weights converge to one of many valid solutions, and the value error, plotted in black on the right side converges to zero. The left plots of Figure 5 show the same evolution of the weights and step-sizes for AMSGrad. AMSGrad is successful in reducing the step-size for the redundant feature, however the step-sizes of the other features decay quickly and then begin growing again preventing convergence to low value error.

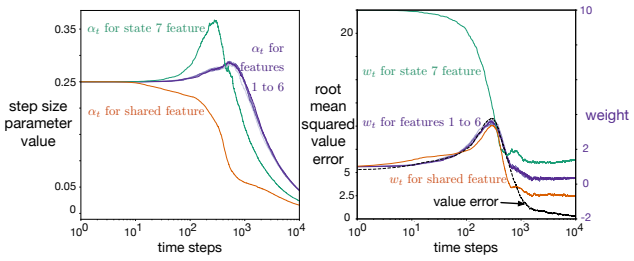

Figure 4: The step-size parameter values over time, and the corresponding weights learned by AdaGain in Baird's counterexample, with results averaged over 1000 independent runs. AdaGain is able to adapt the step-sizes of each feature in such a way that off-policy TD(λ) converges.

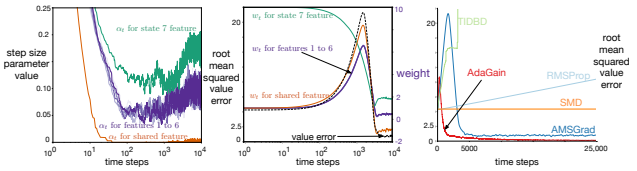

Figure 5: The step-size parameter values over time, and the corresponding weights learned by AMSGrad, and learning curves for several methods in Baird's counterexample. Results averaged over 1000 independent runs. TD(λ) combined with AdaGain achieves the best performance. AMSGrad also prevents divergence, but converges to worse value error.

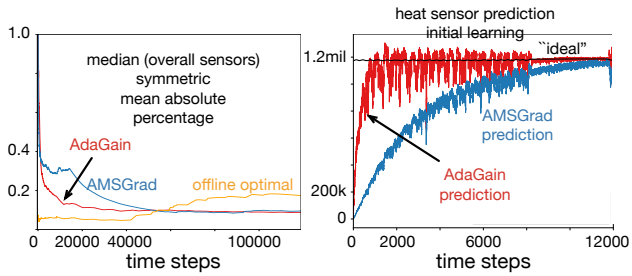

Figure 6: The median symmetric mean absolute percentage error (SMAPE) across all 53 sensors (left), with a plot of the predictions for the heat sensor versus the ideal prediction in early learning (right). The ideal predictions are computed offline using all future data (as described in (Modayil, White, and Sutton 2014)), but the predictions are learned online and incrementally. The learning curve shows that the predictions learned by AdaGain achieve good accuracy more quickly than those learned by AMSGrad. The right plot highlights early learning performance on the heat sensor—from time zero—illustrating that AdaGain's prediction more quickly approaches the desired magnitude and then maintains good stability. This is particularly notable because the heat sensor targets in this case are unnormalized, obtaining values over 1 million. We also include the optimal predictions computed by solving a system of equations offline (again as in (Modayil, White, and Sutton 2014)). The optimal solution makes use of only the first 40,000 data points for each sensor, reflecting the realistic scenario of computing predictions from a limited batch of data, and later using the offline solution for online prediction. As to be expected the SMAPE for these offline optimal predictions is low on the training data (first 40,000 time steps), and much higher on later data.

Experiments on robot data

In our final experiment we recreate nexting (Modayil, White, and Sutton 2014), using TD(λ) to make dozens of predictions about the future values of robot sensor readings. We formulate each prediction as estimating the discounted sum of future sensor readings, treating each sensor as a reward signal with discount factor of $\gamma = 0.9875$ corresponding to approximately 80 second predictions. Using the freely available nexting data set (144,000 samples, corresponding to 3.4 hours of runtime on the robot), we incrementally processed the data on each step constructing a feature vector from the sensor vector, and making one prediction for each sensor. At the end of learning we computed the "ideal" prediction offline and computed the symmetric mean absolute percentage error of each prediction, and aggregated the 50 learning curves using the median. We used the same non-linear coarse recoding of the sensor inputs described in the original work, giving 6065 binary feature components for use as a linear representation.

For this experiment we reduced the number of algorithms, using AMSGrad as the best performing quasi-second order method based on our synthetic task experiments and AdaGain as the representative meta-descent algorithm. The meta

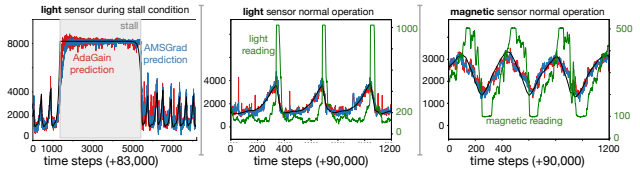

Figure 7: Three snapshots in time of the predictions learned by AdaGain compared with the offline ideal predictions. Each of the three plots highlights a different part of the dataset to give an alternative perspective on the accuracy of AdaGain's learned predictions. The leftmost plot we see a situation where the robot stalled unexpectedly directly in front of a bright light source, saturating the light sensor. Due to this sudden unpredictable event, the predictions of both AdaGain and AMSGrad became incorrect. However, AdaGain more quickly adapts learning to adjust its predictions to reflect the new reality, matching the ideal predictions (black line). Otherwise, these plots show that, in general, AdaGain and AMSGrad can track the ideal prediction similarly.

step-size was optimized for both algorithms.

The learning curves in Figure 6 show a clear advantage for AdaGain in terms of aggregate error over all predictions. Inspecting the predictions of one of the heat sensors reveals why. In early learning, AdaGain much more quickly increases the prediction, to near the ideal prediction, whereas AMSGrad much more slowly reaches this point—over 12000 steps. AdaGain and AMSGrad then both track the the ideal heat prediction similarly, and so obtain similar error for the remainder of learning. This advantage in initial learning is also demonstrated in Figure 7, which depicts predictions on two different sensors. For example, AdaGain adapts the predictions more quickly in reaction to the unexpected stall event, but otherwise AdaGain and AMSGrad obtain similar errors. This result also serves as a sanity check for AdaGain, validating that AdaGain does scale to more realistic problems and remains stable in the face of high levels of noise and high-magnitude prediction targets.

Conclusion

In this work, we proposed a new general meta-descent strategy, to adapt a vector of stepsizes for online, continual prediction problems. We defined a new meta-descent objective, that enables a broader class of incremental updates for the base learner, generalizing beyond work specialized to least-mean squares, temporal difference learning and vanilla stochastic gradient descent updates. We derive a recursive update for the stepsizes, and provide a linear-complexity approximation. In a series of experiments, we highlight that meta-descent strategies are not robust to the shape of the optimization surface. The ability to use AdaGain for generic updates enabled us to overcome this issue, by layering AdaGain on RMSProp, a simple quasi-second order approach. We then shown that, with this modification, meta-descent methods can perform better than the more commonly used quasi-second order updates, adapting more quickly in non-stationary tasks.

References

Almeida, L. B.; Langlois, T.; Amaral, J. D.; and Plakhov, A. 1998. On-line learning in neural networks. In Saad, D., ed., *On-Line Learning in Neural Networks*. New York, NY, USA: Cambridge University Press. chapter Parameter Adaptation in Stochastic Optimization, 111–134.

Amari, S.-i.; Park, H.; and Fukumizu, K. 2000. Adaptive Method of Realizing Natural Gradient Learning for Multilayer Perceptrons. *Neural Computation*.

Andrychowicz, M.; Denil, M.; Gómez, S.; Hoffman, M. W.; Pfau, D.; Schaul, T.; and de Freitas, N. 2016. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*.

Baird, L. 1995. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*. Elsevier. 30–37.

Baydin, A. G.; Cornish, R.; Rubio, D. M.; Schmidt, M.; and Wood, F. 2018. Online Learning Rate Adaptation with Hypergradient Descent. In *International Conference on Learning Representations*.

Benveniste, A.; Metivier, M.; and Priouret, P. 1990. *Adaptive Algorithms and Stochastic Approximations*. Springer.

Bordes, A.; Bottou, L.; and Gallinari, P. 2009. SGD-QN: Careful quasi-Newton stochastic gradient descent. *Journal of Machine Learning Research*.

Dabney, W., and Barto, A. G. 2012. Adaptive step-size for online temporal difference learning. In *AAAI*.

Dabney, W., and Thomas, P. S. 2014. Natural Temporal Difference Learning. In *AAAI Conference on Artificial Intelligence*.

Dabney, W. C. 2014. *Adaptive Step-sizes for Reinforcement Learning*. Ph.D. Dissertation, University of Massachusetts - Amherst.

Duchi, J.; Hazan, E.; and Singer, Y. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*.

Finn, C.; Abbeel, P.; and Levine, S. 2017. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *International Conference on Machine Learning*.

Jacobs, R. 1988. Increased rates of convergence through learning rate adaptation. *Neural Networks*.

Jaeger, H. 2000. Observable Operator Processes and Conditioned Continuation Representations. *Neural Computation*.

Kearney, A.; Veeriah, V.; Travnik, J. B.; Sutton, R. S.; and Pilarski, P. M. 2018. Tidbd: Adapting temporal-difference step-sizes through stochastic meta-descent. *arXiv preprint arXiv:1804.03334*.

Kingma, D. P., and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Machine Learning*.

Littman, M. L.; Sutton, R. S.; and Singh, S. 2001. Predictive representations of state. In *Advances in Neural Information Processing Systems*.

Maei, H. R. 2011. *Gradient temporal-difference learning algorithms*. University of Alberta Edmonton, Alberta.

Mahmood, A. R.; Sutton, R. S.; Degris, T.; and Pilarski, P. M. 2012. Tuning-free step-size adaptation. *ICASSP*.

McMahan, H. B., and Streeter, M. 2010. Adaptive Bound Optimization for Online Convex Optimization. In *International Conference on Learning Representations*.

Meyer, D.; Degenne, R.; Omrane, A.; and Shen, H. 2014. Accelerated gradient temporal difference learning algorithms. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*.

Modayil, J.; White, A.; and Sutton, R. S. 2014. Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior* 22(2):146–160.

Nesterov, Y. 1983. A method of solving a convex programming problem with convergence rate O (1/k2). *Soviet Mathematics and Doklady*.

Pan, Y.; Azer, E. S.; and White, M. 2017. Effective sketching methods for value function approximation. In *Conference on Uncertainty in Artificial Intelligence, Amsterdam, Netherlands*.

Pan, Y.; White, A.; and White, M. 2017. Accelerated Gradient Temporal Difference Learning. In *International Conference on Machine Learning*.

Pearlmutter, B. A. 1994. Fast Exact Multiplication by the Hessian. *dx.doi.org*.

Reddi, S. J.; Kale, S.; and Kumar, S. 2018. On the Convergence of Adam and Beyond. In *International Conference on Learning Representations*.

Roux, N. L.; Schmidt, M.; and Bach, F. R. 2012. A stochastic gradient method with an exponential convergence rate for finite training sets. In *Advances in Neural Information Processing Systems*.

Schaul, T.; Zhang, S.; and LeCun, Y. 2013. No More Pesky Learning Rates. In *International Conference on Artificial Intelligence and Statistics*.

Schraudolph, N.; Yu, J.; and Günter, S. 2007. A stochastic quasi-Newton method for online convex optimization. In *International Conference on Artificial Intelligence and Statistics*.

Schraudolph, N. N. 1999. Local gain adaptation in stochastic gradient descent. *International Conference on Artificial Neural Networks: ICANN '99*.

Spall, J. C. 1992. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control* 37(3):332–341.

Sutton, R. S., and Barto, A. G. 1998. *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st edition.

Sutton, R. S., and Tanner, B. 2004. Temporal-Difference Networks. In *Advances in Neural Information Processing Systems*.

Sutton, R. S.; Modayil, J.; Delp, M.; Degris, T.; Pilarski, P.; White, A.; and Precup, D. 2011. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *International Conference on Autonomous Agents and Multiagent Systems*.

Sutton, R.; Koop, A.; and Silver, D. 2007. On the role of tracking in stationary environments. In *International Conference on Machine Learning*.

Sutton, R. S. 1992a. Gain Adaptation Beats Least Squares? In *Seventh Yale Workshop on Adaptive and Learning Systems*.

Sutton, R. 1992b. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI Conference on Artificial Intelligence*.

Tieleman, T., and Hinton, G. 2012. RmsProp: Divide the gradient by a running average of its recent magnitude. In *COURSERA Neural Networks for Machine Learning*.

Wu, Y.; Ren, M.; Liao, R.; and Grosse, R. B. 2018. Understanding Short-Horizon Bias in Stochastic Meta-Optimization. In *International Conference on Learning Representations*.

Zeiler, M. D. 2012. ADADELTA: An Adaptive Learning Rate Method. *arXiv:1411.4000v2 [cs.LG]*.

Stochastic Meta-Descent algorithm

We recreate the SMD derivation, in our notation, for easier reference.

We compute the gradient of the loss function $\ell(\mathbf{w})$, w.r.t. stepsize. We derive the full quadratic-complexity algorithm to start, and then introduce approximations to obtain a linear-complexity algorithm. For stepsize α_i as the ith element in the vector $\boldsymbol{\alpha}$,

$$\frac{\partial \ell(\mathbf{w}(\boldsymbol{\alpha}))}{\partial \alpha_i} = \sum_j^k \frac{\partial \ell(\mathbf{w}(\boldsymbol{\alpha}))}{\partial w_{t,j}} \frac{\partial w_{t,j}}{\partial \alpha_i}$$

Define the following two vectors, for $w_{t,j}$ the j-th element in vector $w_{t,j}$,

$$\mathbf{g}_{t,j} \stackrel{\text{def}}{=} -\frac{\partial \ell(\mathbf{w}(\boldsymbol{\alpha}))}{\partial w_{t,j}} \in \mathbb{R}^k \qquad \text{the gradient update} \tag{7}$$

$$\boldsymbol{\psi}_{t,i} \stackrel{\text{def}}{=} \frac{\partial \mathbf{w}_t}{\partial \alpha_i} \in \mathbb{R}^k. \tag{8}$$

We can obtain vector $\boldsymbol{\psi}_{t,i}$ recursively as

$$\begin{aligned}
\boldsymbol{\psi}_{t+1,i} &= \frac{\partial(\mathbf{w}_t + \boldsymbol{\alpha} \circ \mathbf{g}_t)}{\partial \alpha_i} = \frac{\partial \mathbf{w}_t}{\partial \alpha_i} + \boldsymbol{\alpha} \circ \frac{\partial \mathbf{g}_t}{\partial \alpha_i} + \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{t,i}(\boldsymbol{\alpha}) \\ \mathbf{0} \end{bmatrix} \\
&= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha} \circ \sum_j \frac{\partial \mathbf{g}_t}{\partial w_{t,j}} \frac{\partial w_{t,j}}{\partial \alpha_i} + \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{t,i}(\boldsymbol{\alpha}) \\ \mathbf{0} \end{bmatrix} \\
&= \boldsymbol{\psi}_{t,i} - \boldsymbol{\alpha} \circ (\mathbf{H}_t \boldsymbol{\psi}_{t,i}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{t,i}(\boldsymbol{\alpha}) \\ \mathbf{0} \end{bmatrix} \\
&= (\mathbf{I} - \text{diag}(\boldsymbol{\alpha})\mathbf{H}_t)\boldsymbol{\psi}_{t,i} + \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{t,i}(\boldsymbol{\alpha}) \\ \mathbf{0} \end{bmatrix}.
\end{aligned}$$

The resulting generic updates for quadratic-complexity SMD, with meta stepsize $\bar{\alpha}$, are

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \exp\left(\bar{\alpha} \boldsymbol{\alpha}_t \circ \boldsymbol{\Psi}_t^\top \mathbf{g}_t\right) \tag{9}$$

$$\text{for } (\boldsymbol{\Psi}_t)_{:,i} = \boldsymbol{\psi}_{t,i} \text{ with } \boldsymbol{\Psi}_t \in \mathbb{R}^{k \times k}$$

$$\mathbf{H}_t = \text{Hessian of } \ell_t \text{ w.r.t. } \mathbf{w}_t.$$

$$\boldsymbol{\psi}_{t+1,i} = (1-\beta)\boldsymbol{\psi}_{t,i} - \beta \boldsymbol{\alpha}_t \circ (\mathbf{H}_t \boldsymbol{\psi}_{t,i}) + \beta \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{t,i} \\ \mathbf{0} \end{bmatrix}$$

$\boldsymbol{\psi}_{0,i} = \mathbf{0}$ and $\boldsymbol{\alpha}_0 = 0.1$ (or some initial value). As with AdaGain, the Hessian-vector product $\mathbf{H}_t \boldsymbol{\psi}_{t,i}$ can be computed efficiently, using R-operators. Here, it is irrelevant, because we maintain the quadratic $\boldsymbol{\Psi}$.

For the linear-complexity algorithm, again we set entries $(\boldsymbol{\psi}_{t,i})_j = 0$ for $i \neq j$. Let $\mathbf{H}_{t,i}$ be the ith column of the Hessian. This results in the simplification

$$\begin{aligned}
\boldsymbol{\psi}_{t+1,i} &= \boldsymbol{\psi}_{t,i} - \boldsymbol{\alpha} \circ \sum_j^k \mathbf{H}_{t,j} (\boldsymbol{\psi}_{t,i})_j + \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{t,i}(\boldsymbol{\alpha}) \\ \mathbf{0} \end{bmatrix} \\
&= \boldsymbol{\psi}_{t,i} - \boldsymbol{\alpha} \circ \mathbf{H}_{t,i} (\boldsymbol{\psi}_{t,i})_i + \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{t,i}(\boldsymbol{\alpha}) \\ \mathbf{0} \end{bmatrix}.
\end{aligned}$$

Further, since we will then assume that $(\boldsymbol{\psi}_{t+1,i})_j = 0$ for $i \neq j$, there is no purpose in computing the full vector $\mathbf{H}_{t,i}(\boldsymbol{\psi}_{t,i})_i$. Instead, we only need to compute the ith entry, i.e., for $\frac{\partial \mathbf{g}_{t,i}(\boldsymbol{\alpha})}{\partial w_{t,i}}$.

We can then instead define $\hat{\psi}_{t,i}$ to be a scalar approximating $\frac{\partial w_{t,i}}{\partial \alpha_i}$, with $\hat{\boldsymbol{\psi}}_t$ the vector of these, and the diagonal of the Hessian

$$\hat{\mathbf{h}}_t \stackrel{\text{def}}{=} \left[\frac{\partial^2 \ell(\mathbf{w}(\boldsymbol{\alpha})}{\partial \mathbf{w}_{t,1}^2}, \ldots, \frac{\partial^2 \ell(\mathbf{w}(\boldsymbol{\alpha})}{\partial \mathbf{w}_{t,k}^2} \right] \tag{10}$$

to define the recursion as $\hat{\boldsymbol{\psi}}_{t+1} \stackrel{\text{def}}{=} \hat{\boldsymbol{\psi}}_t - \boldsymbol{\alpha} \circ \hat{\mathbf{h}}_t \circ \hat{\boldsymbol{\psi}}_t + \mathbf{g}_t(\boldsymbol{\alpha})$, with $\hat{\boldsymbol{\psi}}_0 = \mathbf{0}$. The gradient using this approximation, with off-diagonals zero, is

$$\begin{aligned}
\frac{\partial \ell(\mathbf{w}(\boldsymbol{\alpha}))}{\partial \alpha_i} &= \sum_j^k \frac{\partial \ell(\mathbf{w}(\boldsymbol{\alpha}))}{\partial w_{t,j}} \frac{\partial w_{t,j}}{\partial \alpha_i} \\
&\approx \frac{\partial \ell(\mathbf{w}(\boldsymbol{\alpha}))}{\partial w_{t,i}} \frac{\partial w_{t,i}}{\partial \alpha_i} \\
&= \hat{\psi}_{t,i} \mathbf{g}_{t,i}
\end{aligned}$$

The resulting update to the stepsize is

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \exp\left(\bar{\alpha} \boldsymbol{\alpha}_t \circ \hat{\boldsymbol{\psi}}_t \circ \mathbf{g}_t\right) \tag{11}$$

$$\hat{\boldsymbol{\psi}}_{t+1} = (1-\beta)\hat{\boldsymbol{\psi}}_t - \beta \boldsymbol{\alpha}_t \circ \hat{\mathbf{h}}_t \circ \hat{\boldsymbol{\psi}}_t + \beta \mathbf{g}_t.$$

Difference to original SMD algorithm: Now, surprisingly, the above algorithm differs from the algorithm given for SMD. But, that derivation appears to have a flaw, where the gradients of weights taken w.r.t. to a vector of stepsizes is assumed to be a vector. Rather, with the same off-diagonal approximation we use, it should be a diagonal matrix, and then they would also only get a diagonal Hessian. For completeness, we include their algorithm, which uses a full Hessian-vector product.

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \exp\left(\bar{\alpha} \boldsymbol{\alpha}_t \circ \hat{\boldsymbol{\psi}}_t \circ \mathbf{g}_t\right) \tag{12}$$

$$\hat{\boldsymbol{\psi}}_{t+1} = \hat{\boldsymbol{\psi}}_t - \boldsymbol{\alpha}_t \circ \mathbf{H}_t \hat{\boldsymbol{\psi}}_t + \mathbf{g}_t.$$

Note that a follow-up paper that tested SMD (Wu et al. 2018) uses this update, but does not have an error, because they use a *scalar* step size. In fact, in the SMD paper, if the step size had been a scalar, then their derivation would be correct.

The addition of β: The original SMD algorithm did not use forgetting with β. In our experiments, however, we consider SMD with β—which performs significantly better—since our goal is not to compare directly with SMD, but rather to compare the choice of objectives.

Derivations for AdaGain updates

Consider again the generic update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \boldsymbol{\alpha} \circ \Delta_t \tag{13}$$

where $\Delta_t \in \mathbb{R}^d$ is the update for this step, for weights $\mathbf{w}_t \in \mathbb{R}^d$ and constant vector stepsize $\boldsymbol{\alpha}$ and the operator $\circ$ denotes element-wise multiplication.

Maintaining non-negative stepsizes in AdaGain

One straightforward option to maintain non-negative stepsizes is to define a constraint on the stepsize. We can prevent the stepsize from going below a small threshold ϵ (e.g., $\epsilon = 0.001$), ensuring positive stepsizes. The projection onto this constraint set after each gradient descent step simply involves applying the operator $(\cdot)_\epsilon$, which thresholds any values below $\epsilon > 0$ to ϵ. We experimented with this strategy compared to the mentioned exponential form, and found it performed relatively similarly, but required an extra parameter to tune.

Another option—and the one we use in this work—is to use an exponential form for the stepsize, so that it remains positive. One form, used also by IDBD, is to use $\boldsymbol{\alpha} = \exp(\boldsymbol{\beta})$. The algorithm, with or without an exponential form, remains essentially identical to the thresholded version, because

$$\frac{\frac{1}{2}\partial\|\Delta_t(\boldsymbol{\alpha}(\boldsymbol{\beta}))\|_2^2}{\partial\beta_i} = \Delta_t(\boldsymbol{\alpha}(\boldsymbol{\beta}))\frac{\Delta_t(\boldsymbol{\alpha}(\boldsymbol{\beta}))}{\partial\alpha_i}\frac{\partial\alpha_i}{\partial\beta_i}.$$

Therefore, we can still recursively estimate the gradient with the same approach, regardless of how the stepsize $\boldsymbol{\alpha}$ is constrained. For the thresholded form, we simply use the gradient $\Delta_t(\boldsymbol{\alpha}(\boldsymbol{\beta}))\frac{\Delta_t(\boldsymbol{\alpha})}{\partial\alpha_i}$ and then project (i.e., threshold). For the exponential form, the gradient update for $\boldsymbol{\alpha}$ is simply used within an exponential function, as described below.

Consider directly maintaining $\boldsymbol{\beta}$, which is unconstrained. For the function form $\alpha_i = \exp(\beta_i)$, the partial derivative $\frac{\partial\alpha_i}{\partial\beta_i}$ is simply equal to α_i and so the gradient update includes an additional α_i in front. This can more explicitly be maintained, without an additional variable, by noticing that for gradient $g_i = \alpha_i\Delta_t(\boldsymbol{\alpha}(\boldsymbol{\beta}))\frac{\Delta_t(\boldsymbol{\alpha}(\boldsymbol{\beta}))}{\partial\alpha_i}$ for $\beta_{t,i}$

$$\begin{aligned}\alpha_{t+1,i} &= \exp(\beta_{t+1,i})\\ &= \exp(\beta_{t,i} - \bar{\alpha}g_i)\\ &= \exp(\beta_{t,i})\exp(-\bar{\alpha}g_i)\\ &= \alpha_{t,i}\exp(-\bar{\alpha}g_i)\end{aligned}$$

Therefore, we can still directly maintain $\boldsymbol{\alpha}$. The resulting update to $\boldsymbol{\alpha}$ is simply

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1}\exp\left(-\bar{\alpha}\boldsymbol{\alpha}_t \circ \hat{\boldsymbol{\psi}}_t \circ (\mathbf{G}_t^\top\Delta_t)\right) \tag{14}$$

Other multiplicative updates are also possible. Schraudolph (1999) uses an exponential update, but uses an approximation with a maximum, to avoid the expensive computation of the exponential function. Baydin et al. (2018) uses a similar multiplicative update, but without a maximum.

AdaGain for linear TD

In this section, we derive $\mathbf{g}_t$ for a particular algorithm, namely linear TD. LMS updates can be obtained as special cases, by setting $\gamma = 0$. We then provide a more general update algorithm—which does not require knowledge of the form of the update—in the next section. One advantage of AdaGain is that it is derived generically, allowing extensions to many online algorithms, unlike IDBD, and variants which are derived specifically for the squared TD-error.

We first provide the AdaGain updates for linear TD(λ), and then provide the derivation below. For TD(λ), the update is

$$\begin{aligned}\delta_t &\stackrel{\text{def}}{=} r_{t+1} + \gamma_{t+1}\mathbf{x}_{t+1}^\top\mathbf{w}_t - \mathbf{x}_t^\top\mathbf{w}_t\\ \Delta_t &\stackrel{\text{def}}{=} \delta_t\mathbf{e}_t\end{aligned}$$

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1}\exp(-\bar{\alpha}(\Delta_t^\top\mathbf{e}_t)\boldsymbol{\alpha}_{t-1}\circ\mathbf{d}_t\circ\hat{\boldsymbol{\psi}}_t)\\ \hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_{t,i} + \beta\boldsymbol{\alpha}_t\circ\mathbf{e}_t\circ\mathbf{d}_t\circ\hat{\boldsymbol{\psi}}_t + \beta\Delta_t\end{aligned} \tag{15}$$

where $\boldsymbol{\alpha}_0 = 0.1$, $\hat{\boldsymbol{\psi}}_0 = \mathbf{0}$, $\gamma_t \stackrel{\text{def}}{=} \gamma(S_t, A_t, S_{t+1})$

To derive the update for $\boldsymbol{\alpha}$, we need to compute the gradients of the updates, particularly $(\mathbf{g}_t)_i = \frac{\partial\Delta_{t,i}}{\partial\mathbf{w}_{t,i}}$ or for the full algorithm, the Jacobian $\mathbf{G}$.

$$\begin{aligned}\frac{\partial\Delta_t}{\partial\mathbf{w}_{t,i}} &= \mathbf{e}_t\frac{\partial\delta_t}{\partial\mathbf{w}_{t,i}}\\ &= \mathbf{e}_t\frac{\partial}{\partial\mathbf{w}_{t,i}}(r_{t+1} + \gamma_{t+1}\mathbf{x}_{t+1}^\top\mathbf{w}_t - \mathbf{x}_t^\top\mathbf{w}_t)\\ &= \mathbf{e}_t\left(\gamma_{t+1}\mathbf{x}_{t+1} - \mathbf{x}_t\right)_i.\end{aligned}$$

Letting $\mathbf{d}_t \stackrel{\text{def}}{=} \gamma_{t+1}\mathbf{x}_{t+1} - \mathbf{x}_t$, the Jacobian is $\mathbf{G}_t = \mathbf{e}_t\mathbf{d}_t^\top$ and the diagonal approximation is $\mathbf{g}_t = \mathbf{e}_t \circ \mathbf{d}_t$. Because of the form of the Jacobian, we can actually use it in the update to $\boldsymbol{\alpha}$, though not in computing $\hat{\boldsymbol{\psi}}_t$, if we want to maintain linearity. The quadratic complexity algorithm uses $\mathbf{G}$ as given

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1}\exp(-\bar{\alpha}(\Delta_t^\top\mathbf{d}_t)\boldsymbol{\alpha}_{t-1}\circ(\boldsymbol{\Psi}_t^\top\mathbf{e}_t))\\ \boldsymbol{\psi}_{t,i} &= (1-\beta)\boldsymbol{\psi}_{t-1,i}\\ &\quad + \beta\boldsymbol{\alpha}\circ(\mathbf{e}_{t-1}\mathbf{d}_{t-1}^\top\boldsymbol{\psi}_{t-1,i}) + \beta\begin{bmatrix}\mathbf{0}\\ \Delta_{t-1,i}\\ \mathbf{0}\end{bmatrix}\end{aligned}$$

The linear complexity algorithm uses $\mathbf{g}_t$ to update $\hat{\boldsymbol{\psi}}_t$, giving the stepsize update in (15)

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1}\exp(-\bar{\alpha}(\Delta_t^\top\mathbf{d}_t)\boldsymbol{\alpha}_{t-1}\circ\mathbf{e}_t\circ\hat{\boldsymbol{\psi}}_t)\\ \hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_{t,i} + \beta\boldsymbol{\alpha}_t\circ\mathbf{e}_t\circ\mathbf{d}_t\circ\hat{\boldsymbol{\psi}}_t + \beta\Delta_t\end{aligned}$$

Generic AdaGain algorithm

To avoid requiring knowledge about the algorithm update and its derivatives, we can provide an approximation to the Jacobian-vector product and the diagonal of the Jacobian, using finite differences. As long as the update function for the algorithm can be queried multiple times, this algorithm can be easily applied to any update.

To compute the Jacobian-vector product, we use the fact that this corresponds to a directional derivative. Notice that $\mathbf{G}_t^\top\Delta_t$ corresponds to the vector of directional derivatives for each component (function) in the update Δ_t, in the direction of $\mathbf{u} = \Delta_t$, because the dot-product separates in $\mathbf{G}_{t,1}^\top\mathbf{u}, \ldots, \mathbf{G}_{t,k}^\top\mathbf{u}$. Therefore, for update function $\Delta : \mathbb{R}^k \to \mathbb{R}^k$ (such as the gradient of the loss), we get for small $r = 0.001$,

$$\mathbf{G}_t^\top\Delta_t \approx \frac{\Delta(\mathbf{w}+r\mathbf{u}) - \Delta(\mathbf{w}-r\mathbf{u})}{2r} \tag{16}$$

For the diagonal of the Jacobian, we can again use finite differences. An efficient finite difference computation is proposed within the simultaneous perturbation stochastic approximation algorithm (Spall 1992), which uses a random perturbation vector $\boldsymbol{\epsilon}$ to compute the centered difference $\frac{(\Delta(\mathbf{w}+r\boldsymbol{\epsilon})-\Delta(\mathbf{w}-r\boldsymbol{\epsilon}))_i}{2r\boldsymbol{\epsilon}_i}$. This formula provides an approximation to the gradient of the i entry in the update Δ_t with respect to weight i; when computed for all i, this approximates the diagonal of the Jacobian $\hat{\mathbf{j}}_t$. To avoid additional computation, we can re-use the above difference with perturbation $\mathbf{u}$, rather than a random vector $\boldsymbol{\epsilon}$. To avoid division by zero, if $\mathbf{u}$ contains a zero entry, we threshold the normalization with a small constant 10^{-6} to give

$$\hat{\mathbf{j}}_t \approx \frac{\Delta(\mathbf{w}+r\mathbf{u}) - \Delta(\mathbf{w}-r\mathbf{u}))}{2r} \circ (1/\text{sign}(\mathbf{u})\max(10^{-6}, |\mathbf{u}|)) \tag{17}$$

where division is element-wise. another approach would be to sample a random direction $\boldsymbol{\epsilon}$ for this finite difference and use $\Delta(\mathbf{w}+\boldsymbol{\epsilon}) - \Delta(\mathbf{w})$, divided by the absolute value of each element of $\boldsymbol{\epsilon}$. We found empirically that using the same direction as Δ_t was

actually more effective, and more computationally efficient, so we propose that approach.

Using these approximations, we can compute the update to the stepsize as in Equation (6), repeated here for easy reference

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \exp\left(-\bar{\alpha}\boldsymbol{\alpha}_{t-1} \circ \hat{\boldsymbol{\psi}}_t \circ (\mathbf{G}_t^\top \Delta_t)\right)$$
$$\hat{\boldsymbol{\psi}}_{t+1} = (1-\beta)\hat{\boldsymbol{\psi}}_t + \beta\boldsymbol{\alpha}_t \circ \hat{\mathbf{j}}_t \circ \hat{\boldsymbol{\psi}}_t + \beta\Delta_t.$$

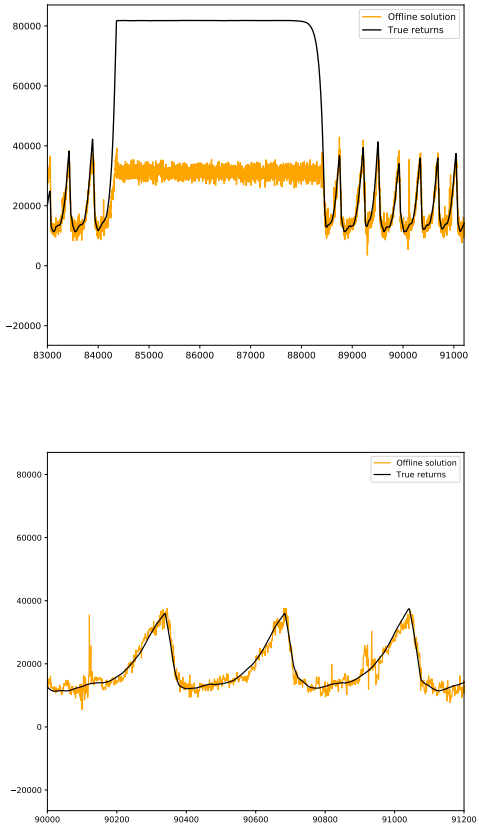

Figure 8: Depicted above are the offline optimal predictions during the light sensor stall, and during the light sensor's normal operation (see Figure 7). The optimal offline solution was trained by computing the linear least-squares solution for the first 40,000 data points, and using that solution to make predictions on the rest of the dataset.